

AD-A160 875 THE DESIGN AND IMPLEMENTATION OF AN OPERATING SYSTEM 1/1
FOR THE IBM PERSONAL COMPUTER(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH A J DEESE DEC 84

THE DESIGN AND IMPLEMENTATION OF AN OPERATING SYSTEM
FOR THE IBM PERSONAL COMPUTER(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH A J DEESE DEC 84

1/1

UNCLASSIFIED

AFIT/CI/NR-85-145T

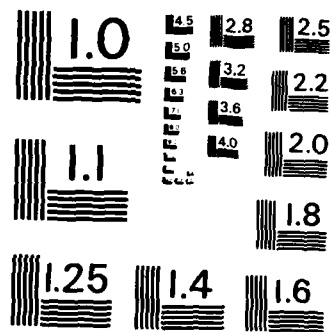
F/G 9/2

NL

END

FIM MED

DTHC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

①
READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER AFIT/CI/NR 85-145T		2. GOVT ACCESSION NO.		3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) The Design And Implementation Of An Operating System For The IBM Personal Computer				5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION	
				6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) Albert James Deese, Jr.				8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Georgia Institute of Technology				10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433 - 6583				12. REPORT DATE Dec 84	
				13. NUMBER OF PAGES 54	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)				15. SECURITY CLASS. (of this report) UNCLASS	
				15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED					
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) B					
18. SUPPLEMENTARY NOTES APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1 LYNN E. WOLAVER 25 APR 85 Dean for Research and Professional Development AFIT, Wright-Patterson AFB OH					
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)					
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) ATTACHED					

AD-A160 875

DTIC FILE COPY

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

85 11 04 076

SUMMARY

This thesis documents the design and implementation of an operating system for the Georgia Institute of Technology Information and Computer Science Laboratory (GIT/ICSL). The operating system, designated PCOS, ~~was~~ developed in order to provide a pedagogical aid which could be used to provide students with a better understanding of operating system principles. PCOS is intended to be a ^(Personal Computer Operating System) simple yet functional operating system which students can analyze, modify, and extend.

PCOS is an acronym which stands for Personal Computer Operating System. PCOS was designed and implemented on an IBM Personal Computer (IBM PC). However, the strategy used to structure PCOS along with the algorithms used to implement PCOS are applicable to most contemporary computer systems.

This thesis presents the requirements and design criteria which were used to guide the design of PCOS. The decisions made during the design of PCOS are discussed. The algorithms and data structures used to implement PCOS are discussed. And, further development of PCOS is also discussed.

FOLD DOWN ON OUTSIDE - SEAL WITH TAPE

AFIT/NR
WRIGHT-PATTERSON AFB OH 45433

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, \$300



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

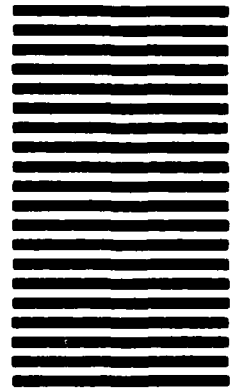
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 73236 WASHINGTON D.C.

POSTAGE WILL BE PAID BY ADDRESSEE

AFIT/ DAA

Wright-Patterson AFB OH 45433



FOLD IN

THE DESIGN AND IMPLEMENTATION OF AN OPERATING SYSTEM
FOR THE IBM PERSONAL COMPUTER

A THESIS

Presented to
the Faculty of the Division of Graduate Studies

By

Albert James Deese, Jr.

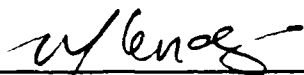
In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Information and Computer Science

Georgia Institute of Technology

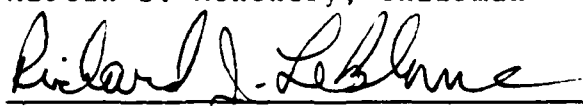
December 1984

The Design and Implementation of an Operating System
for the IBM Personal Computer

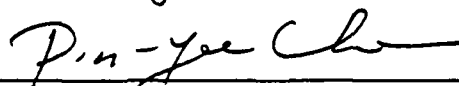
Approved:



Martin S. McKendry, Chairman



Richard J. LeBlanc

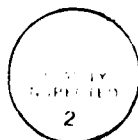


Pin-fee Chen

✓
PER CALL JC

Date approved by Chairman 11-12-84

A-1



ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Dr. Martin S. McKendry for the assistance and guidance he provided over the course of this project. I would also like to thank the members of my reading committee Dr. Richard J. LeBlanc and Dr. Pin-Yee Chen.

I am especially grateful to my wife, Amelia, for her support during the course of this project, and assistance in preparing the final manuscript.

TABLE OF CONTENTS

	Page
LIST OF TABLES	iv
LIST OF ILLUSTRATIONS	v
SUMMARY	vi
Chapter	
I. INTRODUCTION	1
II. THE PCOS PROJECT	4
Requirements	4
Design	4
Implementation	8
III. FURTHER RESEARCH AND DEVELOPMENT	26
IV. CONCLUSION	29
REFERENCES	31
APPENDIX	
A. KERNEL PRIMITIVES	35

LIST OF TABLES

Table	Page
2-1 Kernel Services	11
2-2 Process Control Block	13
2-3 Memory Control Block	16
2-4 Interrupt Vector Status Table Entry	18
2-5 Interprocess Communication Control Block	19
2-6 Device Driver Services	22
2-7 File Server Services	23
2-8 CLI Commands	24
3-1 Proposed Exception Management Services	26

CHAPTER I

INTRODUCTION

Basically, an operating system is a resource manager. It is responsible for the effective and efficient management of computer hardware. Consequently, the operating system is one of the key components of a computer system.

The study of operating systems is an integral part of any progressive computer science curriculum. Introductory courses present the general principles of operating systems while advanced courses present operating system design strategies and implementation techniques in greater detail.

Courses on operating systems should provide insight into the design and implementation of operating systems. However, Lions (1978) points out that the presentation of general principles alone proves to be "rather dry intellectual fodder for students with limited practical experience."

In an effort to provide students with a better understanding of the design and implementation of operating systems, educators such as Lions have proposed several different approaches for teaching operating system

principles. Since students are usually more interested in seeing something work than reading theoretical examples, the approaches emphasize the study and development of actual operating systems. One approach is to conduct a comprehensive study of an actual operating system in an effort to show students how theory has been put into action (Lions, 1978; McCharen, 1980). Another approach is to modify or extend an existing operating system (Bauer, 1975). Yet another approach is to require students to develop a pedagogical operating system from scratch (LaGarde, Olivier & Padiou, 1981; Lane, 1981; Wadland 1980).

This thesis documents the design and implementation of PCOS, an operating system for the Georgia Institute of Technology Information and Computer Science Laboratory (GIT/ICSL). The purpose of this research project is to initiate development of an operating system for the IBM Personal Computer which can be used to provide students with a better understanding of operating system principles. PCOS is intended to be a simple yet functional operating system which students can analyze, modify, and extend. PCOS is an acronym which stands for Personal Computer Operating System.

The next chapter discusses the design and the implementation of PCOS in detail. Chapter Three discusses further development of PCOS. And, Chapter Four presents

conclusions derived from this project.

CHAPTER II

THE PCOS PROJECT

Requirements

As stated in the previous chapter, the purpose of this research project is to develop an operating system which can be used to provide students with a better understanding of operating system principles. The general requirement is that it be a good example of current operating system design practice.

The specific requirements for PCOS were established after reviewing the capabilities of several commercially available operating systems (Banahan & Rutter, 1983; Deitel, 1983; Holt, 1983; Madnick & Donovan, 1974; McKeag, 1976; Zarrella, 1981, 1982). It was decided that PCOS should offer the following capabilities:

- (1) provide a single-user environment,
- (2) support multiprogramming, and
- (3) support sequential disk file organization.

Design

Criteria

Before discussing the design decisions, it is necessary to discuss the general design criteria used to guide this development effort. The general design criteria

are as follows: simplicity, efficiency, and maintainability.

PCOS is intended to be used to provide students with a better understanding of operating system principles. Without simplicity a complete understanding of the system would not be possible. Therefore, PCOS should not contain any unnecessary complexity. It should be based on a small set of relatively simple concepts.

PCOS is also intended to be a simple yet functional operating system. In order to be truly functional it must be efficient.

To be able to adapt to a constantly changing environment, PCOS should be easy to maintain. As a result, PCOS should be well structured.

Decisions

This section discusses the major design decisions made during the development of PCOS. The rationale behind each decision is presented.

First, a computer system was selected for the initial implementation of PCOS. The Information and Computer Science Laboratory at Georgia Tech has several computer systems which are available on a regular basis for student research. However, most of the systems are multi-user systems. Since it would be too costly in terms of lost service alone to dedicate a multi-user system to a single user, they were eliminated from further consideration. In

addition to the multi-user systems, there are several single-user systems. Of the single-user systems, the IBM PC seemed to be the most promising. A typical IBM PC configuration includes a CPU, a keyboard, a monitor, 128k bytes of internal memory, 2 5-1/4" floppy disk drives, and a dot-matrix printer. In addition, the Intel 8088 microprocessor used in the IBM PC supports both segmented memory and interrupts. Given the above considerations, the IBM PC was selected.

Once the implementation system was selected, the implementation language was chosen. The choice of an implementation language was easy. At the time, only a few languages were available: Pascal, Fortran, Basic, and assembly language. Operating systems are often implemented in assembly language for space and time considerations (Freedman, 1977). However, since it is easier to develop and maintain a program written in a high-order language (HOL), Pascal was chosen to be the primary design and implementation language of PCOS. In order to enhance both maintainability and portability, assembly language was used to code only those routines which were neither feasible nor practical to implement using Pascal.

Next the strategy used to structure the PCOS was chosen. Both the monolithic monitor approach and the kernel approach are strategies which can be used to structure operating systems (Deitel, 1983; Holt, 1983).

The monolithic monitor approach collects the resource management functions provided by an operating system into a single, monolithic module. The main advantage of the monolithic monitor approach is its simplicity. All operating system activity takes place in a single module. The main disadvantage is the excessive amount of time external interrupts are disabled. In order to maintain the integrity of the tables it maintains, the monitor disables external interrupts whenever it is running. Since all resource management activity takes place within the monitor, it is possible that interrupts (e.g., information) may be lost.

The kernel approach is an alternative to the monolithic monitor approach. An operating system based on the kernel approach is composed of a set of asynchronous processes and a small executive module (or kernel). Resource management functions are placed in interruptable, asynchronous processes. The kernel provides a set of primitive operations that support the cooperation of asynchronous processes. Using the kernel approach, interrupts are disabled for a shorter period of time. In addition, the operating system is easier to maintain since data structures and algorithms are encapsulated in independent processes. In an effort to create a highly modular and understandable system, the decision was made to use the kernel approach to structure PCOS.

As mentioned in the preceding discussion, the kernel must provide a set of primitive operations that support the cooperation of asynchronous processes. After reviewing the literature generated by past operating system development efforts (Agoston, 1977; Bauer, 1975, 1976; Bayer & Lycklama, 1975; Brinch-Hansen, 1970, 1973; Burgett & O'Neil, 1977; Crowley, 1981; Faro, Messina & Serra, 1981; Frank & Theaker, 1979; Garetti, Laface & Rivoira, 1981; Gorski, 1980; Hammond, 1980; Haridi & Thorelli, 1978; Kahn, 1978; Lycklama & Bayer, 1978; Madnick & Donovan, 1974; Mark, Eggenberger & Nehmer, 1977; Pohjanpalo, 1981; Seidel & Grebe, 1979; Shaw, Weiderman, Andrews, Felcyn, Rieber & Wong, 1975; Sincoskie & Farber, 1980a, 1980b; Solomon & Finkel, 1979; Thorelli, 1978), it was decided that the kernel should provide the following services:

- (1) process management,
- (2) memory management,
- (3) interrupt management,
- (4) interprocess communication management, and
- (5) timer management.

Implementation

System Architecture

PCOS is composed of several hierarchical layers called levels. The levels are designed to be highly independent by encapsulating resources and data

representations within each level. Such encapsulation of objects allows levels to represent abstract views of the objects for which they are responsible. Figure 2-1 depicts the different layers that make up PCOS.

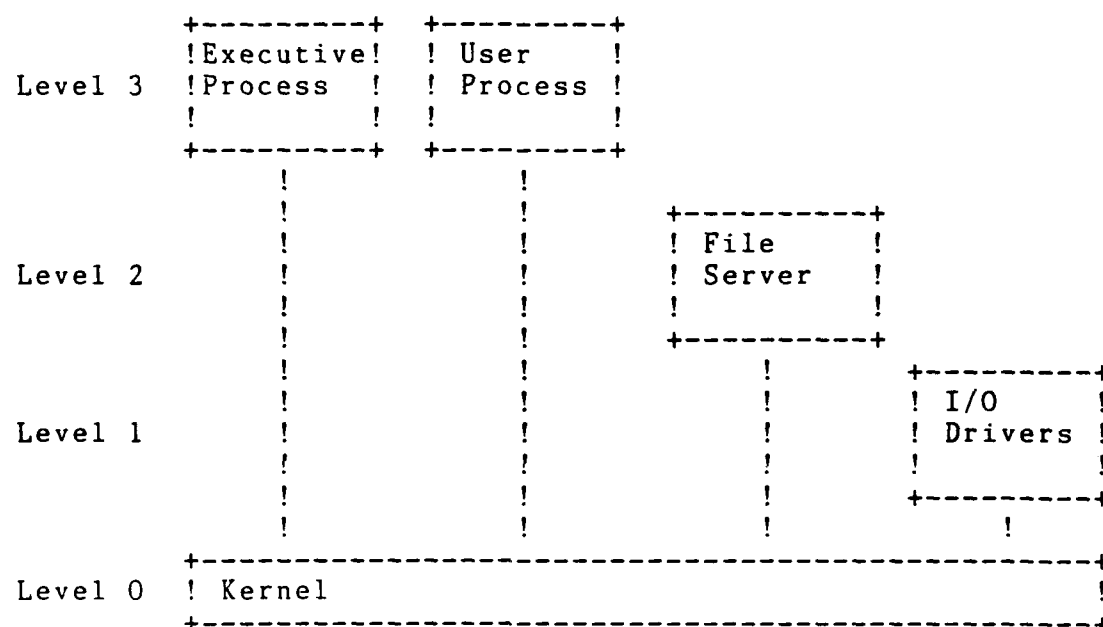


Figure 2-1. PCOS System Hierarchy

The lowest layer, level 0, is the system kernel. The kernel is the nucleus of the operating system. It provides a basic set of services which are available to all processes in the system. These services facilitate process management, memory management, interrupt management, interprocess communication management, timer management, and debugging.

Level 1, the Basic I/O System (BIOS), is composed of a set of processes known as device drivers. These device

drivers are responsible for providing I/O device support. PCOS currently includes a console driver, a disk driver, and a printer driver.

The Extended I/O System (XIOS), level 2, currently includes a single process, the File Server. The File Server presently supports only sequential file access.

The highest layer, level 3, is composed of both user and system processes. Currently, the only processes which reside at level 3 are the Executive Process (EXEC) and the Command Language Interpreter (CLI). EXEC is responsible for starting the system. It does this by first creating and then activating the other processes which compose PCOS. And, the CLI is the process responsible for providing the user interface to PCOS.

Component Details

The Kernel. The kernel provides a set of basic services which can be used by any process in the system. The kernel primitives or services can be grouped into six functional categories: process management, memory management, interrupt management, interprocess communication management, timer management, and debugging. Table 2-1 lists the services provided by the kernel. And, Appendix A contains descriptions of the kernel primitives.

The first category of primitives manipulate processes. Processes are programs that perform a specific function. A process consists of a sequence of

Table 2-1. Kernel Services

CATEGORY	SERVICE
Process Management	Create Process Activate Process Sleep Suspend Process Destroy Process Find Process
Memory Management	Allocate Memory Deallocate Memory
Interrupt Management	Connect to Interrupt Await Interrupt Signal Interrupt Disconnect from Interrupt
Interprocess Communication Management	Send Message Receive Message
Timer Management	Set Clock Read Clock
Debugging	System Dump System Trace

instructions, and a set of resources. Processes can be categorized as either user or system processes. A user process is a process that is written by a user. It performs a function directly for the user. A system process is a process that performs functions (or services) for a user process. System processes are supplied with PCOS that provide basic device management services, and extended I/O services.

Processes can be created and destroyed dynamically.

The maximum number of processes that can exist simultaneously is specified at system generation. The current implementation of PCOS can handle up to 16 concurrent processes. Each process within PCOS has a unique identification number (PID) associated with it.

Each process has a context associated with it. The context of a process is the information that specifies the current status of the process. The current values of the processor registers, including the instruction pointer, and the resources currently allocated to it define the context of the process.

When a process' execution is interrupted, its context is saved in the process control block (PCB) associated with the process so that it can be restarted at a later time. A PCB is a system data structure which is allocated to a process when it is created. A PCB has two major sections: a process descriptor block (PDB); and an interrupt save area (ISA). Table 2-2 describes the contents of a PCB. The PDB contains information including the process identification number, and the priority of the process. The ISA provides a storage area for the process' registers when it is interrupted.

During its existence, a process goes through various process states. Figure 2-2 depicts the various states that a process may go through during its existence.

A process is undefined until its existence is

Table 2-2. Process Control Block

FIELD NAME	DESCRIPTION
PROCESS DESCRIPTOR BLOCK:	
SUCCEEDING_PCB	The id of the next PCB on this queue. This field is used to link PCBs on the ready & delay queues.
PRECEEDING_PCB	The id of the previous PCB on this queue. This field is used to link PCBs on the ready & delay queues.
ID	The PID associated with this PCB.
NAME	A 5 character string which identifies the process which this PCB represents.
PARENT	The PID of the parent process.
YOUNGER_SIB	The PID of the process created by the parent after this process.
OLDER_SIB	The PID of the process created by the parent before this process.
CHILDREN	The PID of the last process created by this process.
PRIORITY	The priority at which this process executes.
STATUS	The current status of this process: running, ready, waiting, suspended, or undefined.
BLOCKS	The events this process is waiting on: message, interrupt, and/or timeout. This field is only meaningful if STATUS is waiting.
WAKEUPS	The events that have happened to this process: message, interrupt, and/or timeout.
MESSAGE_COUNT	The number of messages queued for this process.

Table 2-2. Process Control Block (continued)

FIELD NAME	DESCRIPTION
MESSAGES	A pointer to the first IPCCB (message) for this process.
DELAY	A word which specifies the time, in system time units, the process is to be delayed. This field is only meaningful if STATUS is waiting and BLOCKS is timeout.
NEXT_PCB	The id of the next PCB on either the active or free queue.
PREV_PCB	The id of the previous PCB on either the active or free queue.
INTERRUPT SAVE AREA:	
AX	AX register save area.
BX	BX register save area.
CX	CX register save area.
DX	DX register save area.
SP	SP register save area.
BP	BP register save area.
SI	SI register save area.
DI	DI register save area.
CS	CS register save area.
DS	DS register save area.
SS	SS register save area.
ES	ES register save area.
IP	IP register save area.
FLAGS	FLAGS register save area.

recorded in a PCB. When the information about a process has been entered into a PCB, the process moves to the suspended state. A process remains in the suspended state until it is activated by another process. After it is activated, a process moves to the ready state. A ready process is inserted into a first-in, first-out priority queue known as the ready list. As processes complete execution, the process dispatcher removes the next ready process from the ready list and assigns the processor to it. A process is assigned to the processor by restoring its registers from its ISA and then transferring control to it. A process remains in the running state until it is interrupted, waits for a message, is suspended by another process, or completes its processing.

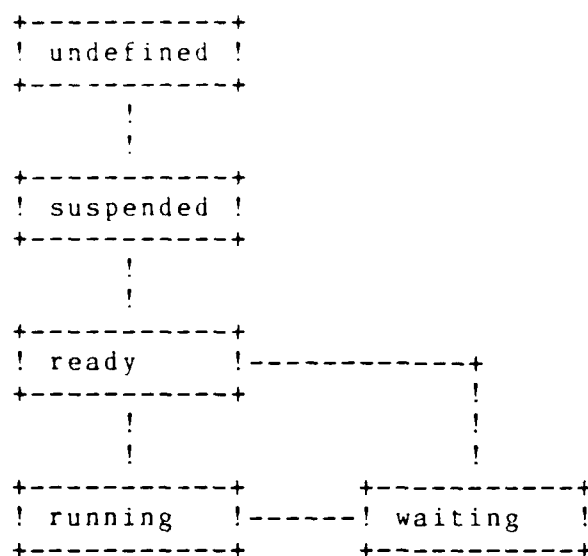


Figure 2-2. Process State Diagram

Six primitives are provided to manage processes: CREATE_PROCESS, ACTIVATE_PROCESS, SLEEP, SUSPEND_PROCESS, DESTROY_PROCESS, and FIND_PROCESS. A process can create a descendant process (child) using the CREATE_PROCESS primitive. First, a PCB is removed from the free PCB queue and encoded with the information supplied in the call. Then the new PCB is queued on the active PCB queue. After creating a child process, the parent can activate it using the ACTIVATE_PROCESS primitive, suspend it using the SUSPEND_PROCESS primitive, or destroy it using the DESTROY_PROCESS primitive. The DESTROY_PROCESS primitive also destroys all descendants of the destroyed process. A process can find a previously created process using the FIND_PROCESS primitive.

The second category of primitives manipulate memory. In the current implementation of PCOS, a fixed amount of memory is linked onto a free memory queue. Table 2-3 describes the format of a memory control block. The memory

Table 2-3. Memory Control Block

FIELD NAME	DESCRIPTION
NEXT	A pointer to the next MCB in the free memory queue.
SIZE	The size, in paragraphs, of this memory block.
FILLER	Forces size of MCB to be 16 bytes (1 paragraph).

linked on the free memory queue is used to satisfy requests for memory from processes.

Two primitives are provided to manage memory: `ALLOCATE_MEMORY`, and `DEALLOCATE_MEMORY`. The `ALLOCATE_MEMORY` primitive searches the free memory queue for the first block of memory that satisfies the request. If the amount of memory found is larger than the amount of memory requested, it is split and the excess returned to the free memory queue. The `DEALLOCATE_MEMORY` primitive returns the specified block of memory to the free memory queue. If possible, the returned block is combined with adjacent memory blocks on queue to form one large block of free memory.

The third category of primitives manipulate interrupts. To redirect control after an interrupt occurs, the Intel 8088 microprocessor uses an Interrupt Vector Table (IVT). The IVT starts at location 0 of main memory and contains 256 interrupt vectors which are numbered 0 - 255. Each interrupt vector can be loaded with the address of the interrupt-service routine that handles that type of interrupt.

In order to manage the IVT, the kernel maintains an Interrupt Vector Status Table (IVST). Table 2-4 describes the contents of an entry in the IVST. There is one entry in the IVST for each interrupt vector in the IVT.

During system initialization, interrupt vectors

Table 2-4. Interrupt Vector Status Table Entry

! FIELD NAME !	! DESCRIPTION !
! AVAILABLE !	! The status of this interrupt: allocated, or unallocated (e.g. available). !
! PID !	! The PID of the interrupt handler process associated with this interrupt. !

0 - 254 are marked as available for use, and interrupt vector 255 is marked as unavailable. Interrupt vector 255 is the interrupt vector used by processes to access the kernel.

Four primitives are provided to manage interrupts: `CONNECT_INTERRUPT`, `AWAIT_INTERRUPT`, `SIGNAL_INTERRUPT`, and `DISCONNECT_INTERRUPT`. The `CONNECT_INTERRUPT` primitive reserves an interrupt vector for a process and assigns an interrupt handler to the interrupt vector. The process can then use the `AWAIT_INTERRUPT` primitive to suspend its execution until its interrupt handler uses the `SIGNAL_INTERRUPT` primitive to activate it or a specified time elapses. The `DISCONNECT_INTERRUPT` primitive cancels the assignment of an interrupt vector to a process.

The fourth category of primitives manipulate messages. Processes communicate with one another by exchanging both commands and data. Processes exchange both commands and data through the use of messages.

Messages are the basic unit of information exchange

between processes. The interprocess communication (IPC) facility of PCOS provides a flexible communication mechanism which can be used to support communication between processes, general network communication, and resource sharing. The interprocess communication facility provided by PCOS is a pure datagram facility. It neither guarantees delivery of a message nor acknowledges its receipt.

A message is delivered using an interprocess communication control block (IPCCB). A IPCCB consists of two parts: (1) a header; and (2) a body. The format of a IPCCB is illustrated in Table 2-5. The message header contains information which identifies the the process sending the message, and the length of the body of the message. The body of the message contains the information being exchanged.

Table 2-5. Interprocess Communication Control Block

! FIELD NAME !	! DESCRIPTION !
! NEXT	! A pointer to the next IPCCB in this process' message queue.
! SOURCE	! The PID of the process which sent this message.
! LENGTH	! The length, in bytes, of this message.
! DATA	! The message.

Two primitives are provided to manage messages:

SEND_MESSAGE, and RECEIVE_MESSAGE. The SEND_MESSAGE primitive sends the message specified to the designated process. After the kernel allocates memory for an IPCCB for the message, the message along with its length and the PID of the sending process is copied into the IPCCB. The IPCCB is then linked to the FIFO message queue of the destination process. If the destination process is awaiting a message and executes at a higher priority than the sending process, it receives control of the CPU and the sending process is inserted into the ready list. If the destination process is awaiting a message and executes at a priority equal to or lower than the sending process, it is inserted into the ready list and the sending process continues execution. And, if the destination process is not awaiting a message, the sending process simply continues execution.

The RECEIVE_MESSAGE primitive returns the message at the head of the process' FIFO message queue. The message is first copied into the process' data space, then dequeued from the message queue, and finally the memory used to buffer the message is returned to the system. If no messages are queued and no delay is specified, control returns to the receiving process; however, if a delay is specified, the process doesn't receive control until it receives a message or the delay has elapsed. And, if an infinite delay is specified, the

process doesn't resume execution until it receives a message.

The fifth category of primitives manipulate the system clock. The system clock is initialized to 00:00:00 hours 0 ticks during system initialization. A system time unit or tick is 1/20th of a second in duration.

Two primitives are provided to manage the system clock: SET_CLOCK and READ_CLOCK. The SET_CLOCK primitive sets the system clock to the value specified; while the READ_CLOCK primitive returns the current setting of the system clock.

And, the sixth category of primitives facilitate debugging. The primitives which facilitate debugging are SYSTEM_DUMP and SYSTEM_TRACE. The SYSTEM_DUMP primitive produces a hexadecimal dump of the contents of the registers and the specified memory block at the time of the dump on the system printer. The SYSTEM_TRACE primitive prints the message specified on the system printer.

The Basic I/O System. The Basic I/O System (BIOS) of PCOS is responsible for device management. Three device drivers are included in the BIOS of PCOS: a console driver, a disk driver, and a printer driver. The services provided by each of the driver processes are listed in Table 2-6.

Communication with the device driver processes that form the BIOS of PCOS is achieved through the use of

Table 2-6. Device Driver Services

DEVICE DRIVER	SERVICE
Console	Read a Character from Keyboard Write a Character to Video Monitor
Disk	Read a Sector Write a Sector Reset Diskette System Verify Sector Format Disk
Printer	Write Character to Printer Reset Printer Read Printer Status

messages. Basically, each device driver is a server process that is responsible for a specific system resource. Each server process was developed using the Requestor-Server Model (Britton & Stickel, 1980) as a guide. Figure 2-3 illustrates the basic form of a server process.

```

procedure terminal;
begin
  loop
    receive_a_message;
    case message.type of
      when 0 => read_character;
      when 1 => display_character;
    end case;
    send_a_reply;
  end loop;
end terminal;

```

Figure 2-3. Example Server Process

Each server process (e.g., console device driver)

waits for a service request using the RECEIVE_MESSAGE primitive. Once it has processed a service request, it sends a reply message to the requestor using the SEND_MESSAGE primitive. This sequence of operations continues until the server is terminated.

The Extended I/O System. The Extended I/O System (XIOS) sequential disk file access. The File Server is the only component of the XIOS. Table 2-7 lists the services that are provided by the File Server.

Table 2-7. File Server Services

CATEGORY	SERVICE
File System Volume	Mount a File System Volume Dismount a File System Volume
File Connection	Open File Close File
File I/O	Read a Character from a File

Like the device drivers that form the BIOS of PCOS, the file server is also a server process. Other processes can communicate with the file server through the use of messages.

To open a file, a task sends an OPEN message which contains the name of the file to be opened to the file server. The task then blocks itself by waiting for the file server's reply. When the file server receives the

message, it interprets it, opens the file (if it exists), and assigns a unique file id to it. The file server then sends a completion message which contains the file id to the process. This causes the process to be rescheduled. Using the file id assigned by the file server, the process can now perform I/O operations on the file.

The Command Language Interpreter. The system operator or user interacts with PCOS through the command language interpreter (CLI). The user issues commands to the system through a command language which is interpreted by the CLI.

Table 2-8. CLI Commands

COMMAND	DESCRIPTION
DISMOUNT	Mounts a file system volume on disk drive 0.
DISPLAY	Displays the contents of a file in hexadecimal on either the system console or the printer.
MOUNT	Dismounts a file system volume from disk drive 0.
PRINT	Prints the contents of a file on the printer.
TIME	Displays the current time.
TYPE	Displays the contents of a file on the system console.

The CLI prompts the user for a command line by

writing '\$ ' to the system console. A command line consists of a command name terminated by a carriage return. Table 2-8 lists the commands which are provided by the CLI. Once the user enters a command line, the CLI checks to see if the command is valid. If it is, the command is then executed; if it isn't, the CLI displays an error message and then prompts the user for another command line. The user is prompted for any arguments that a command needs to perform its function.

When a command has completed execution, the CLI prompts the user for another command line.

The Executive Process. The initialization or startup of both system and user processes is accomplished by the Executive Process (EXEC). After all system data structures have been initialized by the kernel initialization routine, control is passed to EXEC. EXEC then creates and activates the other processes that make up PCOS using the CREATE_PROCESS and ACTIVATE_PROCESS kernel primitives. Once it has started the other system processes, EXEC suspends execution awaiting a system termination message.

CHAPTER III

FURTHER RESEARCH AND DEVELOPMENT

The need for the following modifications to the PCOS kernel was made apparent during the development of the BIOS device driver processes.

The current implementation of the PCOS kernel does not include primitive operations for exception management. An exception occurs whenever a kernel primitive fails. Each kernel primitive returns a condition code that indicates the success or failure of an operation. Currently, processes must check the condition code after each kernel call in order to determine if an exception occurred. In order to reduce the amount of checking a process must perform, the kernel should be modified to include exception management primitives.

Table 3-1. Proposed Exception Management Services

CATEGORY	SERVICE
Exception Management	Create Exception Handler Destroy Exception Handler

Processes should be able to specify an exception handler that is to receive control whenever an exception

occurs. Table 3-1 lists the exception management services that the kernel should provide.

Another operation which the PCOS kernel does not currently support is the selective receipt of messages. This seems to be the greatest deficiency of the PCOS kernel. Consider the following scenario. Process A sends a message requesting some service to Process B, and then continues processing. In order to fulfill the request, Process B sends a request message to Process C and then waits for a response message from Process C. Before Process C can respond, Process A issues another request message to Process B. Since Process B is waiting for a message, it is awakened and given the second request message from Process A. However, Process B is expecting a response message from Process C, not a request message from Process A.

There are two possible solutions to this problem. The first solution is to require processes internally queue messages that they are not ready to process. The second solution is to modify the kernel to include a selective message receipt primitive. The second solution is recommended since it simplifies the construction of server processes.

This solution can be implemented by modifying the `receive_message` primitive to only receive a message from a specified process, and by adding a `receive_any_message`

primitive that receives a message from any process.

Once the above modifications are made, the kernel will be suitable for continued development of PCOS.

CHAPTER IV

CONCLUSION

This thesis documents the design and implementation of an operating system for the IBM PC. The purpose of this research project was to develop an operating system which could be used to provide students with a better understanding of operating system principles.

After discussing the motivation for the PCOS project, the development of PCOS was discussed. First, the requirements for PCOS were discussed. The requirements for PCOS were established after reviewing the capabilities offered by several commercially available operating systems.

Next, the design criteria and the overall design of PCOS was discussed. The design of PCOS is based on the kernel approach. New user requirements such as the need to support a new I/O device can be easily supported through the addition of new processes. In retrospect, we believe the kernel-based design of PCOS provides a good base for further research.

Then, the implementation of PCOS was discussed. PCOS was designed and implemented on an IBM PC. Because of the work needed to construct a complete operating system,

this project focused on the design and implementation of the kernel and the low-level device drivers. The kernel provides a set of primitive operations that support cooperation of asynchronous processes. And, the device drivers provide support for a video display, keyboard, disk drive, and printer. A simplistic file server and command language interpreter were also developed.

Modifications and extensions to PCOS were then presented. The modifications presented are necessary in order to refine the current implementation of PCOS on the IBM PC. The full implementation of the File Server, and the Command language Interpreter has been left for future research efforts.

The author feels that PCOS is a valuable pedagogical aid. Although PCOS is not nearly as complex as OS/MVS or Unix, it does contain many of the basic concepts found in commercially available operating systems. Due to its more simplistic nature, a detailed description is easier to provide and thus easier for students to understand.

This research project incorporated the use of various techniques in several topic areas and resulted in a challenging and rewarding experience for the author. The author hopes that this project will generate continued interest in the subject of operating system design.

REFERENCES

- Agoston, Max K. "A Microprocessor Operating System: The Kernel." Dr. Dobb's Journal, 2 (September 1977), 20-40.
- Banahan, Mike, and Andy Rutter. The Unix Book. New York: John Wiley & Sons, Inc., 1983.
- Bauer, Henry R. "The Design of a TI980A Operating System for Classroom Use." ACM SIGCSE Bulletin, 7, No. 1, (February 1975), 20-22.
- Bauer, H. R., G. D. Thomas, and J. Van Baalen. "A Multiprogramming Operating System for the TI980A." Proceedings of ACM 76, 1976, Houston, Texas, pp. 247 - 251.
- Bayer, D. L., and H. Lycklama. "MERT - A Multi-Environment Real-Time Operating System." Proceedings of the Fifth Symposium on Operating Systems Principles, 19-21 November 1975, Austin, Texas, pp. 33-42.
- Brinch-Hansen, Per. "The Nucleus of a Multiprogramming System." Communications of the ACM, 13, No. 4 (April 1970), 238-241, 250.
- Brinch-Hansen, Per. Operating System Principles. Englewood Cliffs, N. J.: Prentice-Hall, 1973.
- Britton, Dianne E., and Mark E. Stickel. "An Interprocess Communication Facility for Distributed Applications." IEEE COMPCON 80, 23-25 September 1980, Washington, D.C., pp. 590-595.
- Burgett, Kenneth, and Edward F. O'Neil. "An Integral Real-Time Executive for Microcomputers." Computer Design, 16, No. 7 (July 1977), 77-82.
- Crowley, Charles. "The Design and Implementation of a New UNIX Kernel." AFIPS Conference Proceedings, Vol. 50. 4-7 May 1981, Chicago, Illinois, pp. 265-271.
- Deitel, Harvey M. An Introduction to Operating Systems. The Systems Programming Series. Reading, Ma.: Addison-Wesley, 1983.

- Faro, A., G. Messina, and A. Serra. "A Network Operating System for Local Computer Networks." Proceedings of the IEEE Real-Time Systems Symposium. 8-10 December 1981, Miami Beach, Florida, pp. 13-21.
- Frank, G. R., and C. J. Theaker. "The Design of the MUSS Operating System." Software-Practice and Experience, 9, No. 8 (August 1979), 599-620. (a)
- Frank, G. R., and C. J. Theaker. "MUSS - The User Interface." Software-Practice and Experience, 9, No. 8 (August 1979), 621-631. (b)
- Freedman, A. L., and R. A. Lees. Real-time Computer Systems. Computer Systems Engineering Series. New York: Crane, Russak & Company, Inc., 1977.
- Garetti, P., P. Laface, and S. Rivoira. "On the Development of a Distributed Operating System Kernel for Real-Time Applications." Proceedings of the IFAC/IFIP Workshop on Real Time Programming. 31 August-2 September 1981, Kyoto, Japan, pp. 107-114.
- Gorski, J. "KRTM -- A Kernel Which Supports Real Time Multiprograms." Proceedings of the IFAC/IFIP Workshop on Real Time Programming. 14-16 April 1980, Schloss Retzhof, Leibnitz, Austria, pp. 9-18.
- Hammond, Richard A. "Experiences with the Series/1 Distributed System." IEEE COMPCON 80. 23-25 September 1980, Washington, D. C., pp. 585-589.
- Haridi, S., and Lars-Erik Thorelli. "Design and Use of a Simple Monitor for Small Computers." Proceedings of the IFAC/IFIP Workshop on Real Time Programming. 19-21 June 1978, Mariehamn/Åland, Finland, pp. 35-40.
- Holt, R. C. Concurrent Euclid, The Unix System, and Tunis. Addison-Wesley Series in Computer Science. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1983.
- Kahn, Kevin C. "A Small-Scale Operating System Foundation for Microprocessor Applications." Proceedings of the IEEE, 66, No. 2 (February 1978), 209-216.
- LaGarde, J. M., G. Olivier, and G. Padiou. "An Operating System Course Project." ACM SIGCSE Bulletin, 13, No. 2 (June 1981), 34-48.

- Lane, Malcolm G. "Teaching Operating Systems and Machine Architecture -- More on the Hands-on Laboratory Approach." ACM SIGCSE Bulletin, 13, No. 1 (February 1981), 28-36.
- Lions, J. "An Operating System Case Study." Operating Systems Review, 12, No. 3 (July 1978), 46-53.
- Lycklama, H., and D. L. Bayer. "UNIX Time-Sharing System: The MERT Operating System." Bell System Technical Journal, 57, No. 6 (July-August 1978), 2049-2086.
- Madnick, S. E., and J. J. Donovan. Operating Systems. New York: McGraw-Hill, 1974.
- Mark, Anthony, Otto Eggenberger, and Jurgen Nehmer. "Experiences in the Design and Implementation of a Structured Real-Time Operating System." Proceedings of the IFAC/IFIP Workshop on Real-Time Programming, Eindhoven, Netherlands, 20-22 June 1977, 133-137.
- McCharen, Edith A. "MVS in the Classroom." ACM SIGCSE Bulletin, 12, No. 1 (February 1980), 81-82.
- McKeag, R. M. Studies in Operating Systems. New York: Academic Press, 1976.
- Pohjanpalo, Hannu. "MROS -- 68K, a Memory Resident Operating System, for MC68000." Software-Practice and Experience, 11, No. 8 (August 1981), 845-852.
- Seidel, H. A., and G. Grebe. "A Kernel for a Concurrent Pascal Operating System." Operating Systems: Theory and Practice. D. Lanciaux (ed.). North-Holland Publishing Company, 1979.
- Shaw, Alan. "A Multiprogramming Nucleus with Dynamic Resource Facilities." Software-Practice and Experience, 5, No. 3 (July-September 1975), 245-267.
- Sincoskie, W. David, and David J. Farber. "SODS/OS: A Distributed Operating System for the IBM Series/1." Operating Systems Review, 14, No. 3 (July 1980), 46-55.
- Soloman, Marvin H., and Raphael A. Finkel. "The Roscoe Distributed Operating System." Proceedings of the Seventh Symposium on Operating System Principles, Pacific Grove, California, 10-12 December 1979, 108-114.

- Thorelli, Lars-Erik. "A Monitor for Small Computers."
Software-Practice and Experience. 8, No. 4 (July-
August 1978), 439-450.
- Wadland, Kenneth R. "Operating System Projects for
Undergraduates." ACM SIGCSE Bulletin, 12, No. 1
(February 1980), 75-80.
- Zarrella, John, ed. Microprocessor Operating Systems.
Suisun City, California: Microcomputer Applications,
1981.
- Zarrella, John, ed. Microprocessor Operating Systems.
Vol. 2. Suisun City, California: Microcomputer
Applications, 1982.

APPENDIX A

KERNEL PRIMITIVES

ACTIVATE_PROCESS

FORMAT:

```
status := activate_process(process);
```

INPUT PARAMETERS:

process	A word containing the id of the process to be activated.
---------	--

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The ACTIVATE_PROCESS primitive activates a process if it is suspended.

CONDITION CODE:

CC_OK	No exceptional conditions.
CC_EXIST	The specified process does not exist.
CC_ACTIVE	The specified process is already active.

ALLOCATE_MEMORY

FORMAT:

```
status := allocate_memory(amount,  
                           memory);
```

INPUT PARAMETERS:

amount	A word that specifies the number of paragraphs (a paragraph is 16 bytes) requested.
--------	---

OUTPUT PARAMETERS:

memory	A pointer in which PCOS will return the address of the first available byte of the allocated memory block.
status	A word which contains the condition code generated by this primitive.

DESCRIPTION:

The ALLOCATE_MEMORY primitive returns a pointer to the first available byte of the requested memory block.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_MEMORY	There is not enough memory available to satisfy the request.

AWAIT_INTERRUPT

FORMAT:

```
status := await_interrupt(delay);
```

INPUT PARAMETERS:

delay	A word which specifies the amount of time the process is willing to wait for an interrupt. If zero, the process is willing to wait indefinitely. If positive, delay indicates the number of system time units the process is willing to wait. There are 20 system time units per second.
-------	--

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The AWAIT_INTERRUPT primitive causes the currently executing process to be suspended until the interrupt with which it is associated occurs or the delay is exhausted.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_TIMEOUT	A timeout occurred.

CONNECT_INTERRUPT

FORMAT:

```
status := connect_interrupt(interrupt,  
                             handler );
```

INPUT PARAMETERS:

interrupt A word indicating the interrupt vector with which the process is to be associated.

handler A pointer to the first instruction of the interrupt handler.

OUTPUT PARAMETERS:

status A word which contains the condition code generated by this primitive.

DESCRIPTION:

The CONNECT_INTERRUPT primitive assigns a process and an interrupt handler to an interrupt vector.

CONDITION CODES:

CC_OK No exceptional conditions.

CC_EXIST The specified interrupt vector does not exist.

CC_ASSIGN The specified interrupt vector is already assigned an interrupt handler.

CREATE_PROCESS

FORMAT:

```
status := create_process(process,  
                           name,  
                           priority,  
                           start_address,  
                           stack_address,  
                           stack_size );
```

INPUT PARAMETERS:

name	A field which contains a string of six ASCII characters giving the name of the process.
priority	A word that specifies the priority of the new process.
start_address	A pointer to the first instructions of the new process.
stack_address	A pointer to the base of the new process' stack.
stack_size	A word containing the size, in bytes, of the new process' stack.

OUTPUT PARAMETERS:

process	A word in which PCOS will return the identification number for the new process.
status	A word which contains the condition code generated by this primitive.

DESCRIPTION:

The CREATE_PROCESS primitive creates a process and returns an id for it.

CONDITION CODES:

CC_OK	No exceptional conditions.
-------	----------------------------

CC_LIMIT

The new process would except the
maximum number of process allowed by
the system.

DEALLOCATE_MEMORY

FORMAT:

```
status := deallocate_memory(memory);
```

INPUT PARAMETERS:

memory	A pointer to the first byte of the memory block to be returned to the system.
--------	---

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The `RELEASE_MEMORY` primitive returns a block of memory to the system.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_EXIST	The value contained in memory does not point to a valid memory block.

DESTROY_PROCESS

FORMAT:

```
status := destroy_process(process);
```

INPUT PARAMETERS:

process	A word containing the id of the process to be destroyed.
---------	--

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The DESTROY_PROCESS primitive deletes the specified process from the system. The process must be suspended before it can be destroyed.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_EXIST	The specified process does not exist.
CC_STATE	The specified process is not suspended.

DISCONNECT_INTERRUPT

FORMAT:

```
status := disconnect_interrupt(interrupt);
```

INPUT PARAMETERS:

interrupt	A word specifying the interrupt vector from which the process is to be disconnected.
-----------	--

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The DISCONNECT_INTERRUPT primitive cancels the assignment of the process to an interrupt vector.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_EXIST	The specified interrupt vector does not exist.
CC_ASSIGN	The specified interrupt vector is not currently assigned an interrupt handler, or is not assigned to the process.

FIND_PROCESS

FORMAT:

```
status := find_process(process,  
                        name  );
```

INPUT PARAMETERS:

name	A field which contains a string of six ASCII characters giving the name of the process.
------	---

OUTPUT PARAMETERS:

process	A word in which PCOS will return the id of the process whose name is identical to the identifier contained in name.
status	A word which contains the condition code generated by this primitive.

DESCRIPTION:

The FIND_PROCESS primitive searches the queue of process known to PCOS. If a process is found whose name is identical to the process name contained in name its id is returned in process. Otherwise, an CC_EXIST exceptional condition occurs.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_EXIST	The specified process does not exist.

READ_CLOCK

FORMAT:

```
status := read_clock(hours,  
                      minutes,  
                      seconds,  
                      ticks);
```

INPUT PARAMETERS:

none

OUTPUT PARAMETERS:

hours	A word containing the present hour.
minutes	A word containing the present minute.
seconds	A word containing the present second.
ticks	A word containing the present tick.
status	A word which contains the condition code generated by this primitive.

DESCRIPTION:

The READ_CLOCK primitive returns the current setting of the system clock.

CONDITION CODES:

CC_OK	No exceptional conditions.
-------	----------------------------

RECEIVE_MESSAGE

FORMAT:

```
status := receive_message(source,  
                           message,  
                           size,  
                           delay );
```

INPUT PARAMETERS:

message	A message buffer.
size	A word containing the size of the message buffer.
delay	A word which specifies the amount of time the process is willing to wait for a message.

OUTPUT PARAMETERS:

source	A word containing the process id of the process that sent the message.
status	A word which contains the condition code generated by this primitive.

DESCRIPTION:

The RECEIVE_MESSAGE primitive returns a message to the calling process.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_TIMEOUT	A message was not received before the delay was exhausted.

SEND_MESSAGE

FORMAT:

```
status := send_message(destination,  
                        message,  
                        size    );
```

INPUT PARAMETERS:

destination	A word containing the id of the process to which the message is to be sent.
message	A message buffer.
size	A word containing the size of the message buffer.

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The SEND_MESSAGE primitive sends a message to the specified process.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_EXIST	The specified process does not exist.

SET_CLOCK

FORMAT:

```
status := set_clock(hours,  
                    minutes,  
                    seconds,  
                    ticks);
```

INPUT PARAMETERS:

hours	A word containing the new hour value.
minutes	A word containing the new minute value.
seconds	A word containing the new second value.
ticks	A word containing the new tick value.

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The SET_CLOCK primitive sets the sytem clock.

CONDITION CODES:

CC_OK	No exceptional conditions.
-------	----------------------------

SIGNAL_INTERRUPT

FORMAT:

```
status := signal_interrupt(interrupt);
```

INPUT PARAMETERS:

interrupt	A word indicating the interrupt vector whose process isto be signaled.
-----------	--

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The SIGNAL_INTERRUPT primitive allows an interrupt handler to activate its associated interrupt process

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_EXIST	The specified interrupt vector does not exist.
CC_ASSIGN	The specified interrupt vector is not assigned an interrupt process.

SLEEP

FORMAT:

```
status := sleep(delay);
```

INPUT PARAMETERS:

delay	A word which specifies the number of system time units the process wishes to be asleep. There are 20 system time units per second.
-------	--

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The SLEEP primitive causes the currently executing process to suspend its execution for a specified amount of time.

CONDITION CODES:

CC_OK	No exceptional conditions.
-------	----------------------------

SUSPEND_PROCESS

FORMAT:

```
status := suspend_process(process);
```

INPUT PARAMETERS:

process	A word containing the id of the process to be suspended.
---------	--

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The SUSPEND_PROCESS primitive suspends a process.

CONDITION CODES:

CC_OK	No exceptional conditions.
CC_EXIST	The process indicated could not be found.

SYSTEM_DUMP

FORMAT:

```
status := system_dump(start_address,  
                        stop_address );
```

INPUT PARAMETERS:

start_address	A pointer containing the address of the first byte to be displayed.
stop_address	A pointer containing the address of the last byte to be displayed.

OUTPUT PARAMETERS:

status	A word which contains the condition code generated by this primitive.
--------	---

DESCRIPTION:

The SYSTEM_DUMP primitive produces a snapshot of the contents of the registers and the specified memory block. The dump is displayed on the system printer.

CONDITION CODES:

CC_OK	No exceptional conditions.
-------	----------------------------

SYSTEM_TRACE

FORMAT:

```
status := system_trace(message);
```

INPUT PARAMETERS:

message A character string.

OUTPUT PARAMETERS:

status A word which contains the condition
code generated by this primitive.

DESCRIPTION:

The SYSTEM_TRACE primitive displays the specified message on the system printer.

CONDITION CODES:

CC_OK No exceptional conditions.

END

FILMED

12-85

DTIC